# A practical index for approximate dictionary matching with few mismatches

Aleksander Cisłak[†] and Szymon Grabowski[‡]

[†] Warsaw University of Technology, Faculty of Mathematics and Information Science, ul. Koszykowa 75, 00–662 Warsaw, Poland, a.cislak@mini.pw.edu.pl
[‡] Lodz University of Technology, Institute of Applied Computer Science, Al. Politechniki 11, 90–924 Łódź, Poland, sgrabow@kis.p.lodz.pl

**Abstract.** Approximate dictionary matching is a classic string matching problem (checking if a query string occurs in a collection of strings) with applications in, e.g., spellchecking, online catalogs, geolocation, and web searchers. We present a surprisingly simple solution called a *split index*, which is based on the Dirichlet principle, for matching a keyword with few mismatches, and experimentally show that it offers competitive space-time tradeoffs. Our implementation in the C++ language is focused mostly on data compaction, which is beneficial for the search speed (e.g., by being cache friendly). We compare our solution with other algorithms and we show that it performs better for the Hamming distance. Query times in the order of 1 microsecond were reported for one mismatch for the dictionary size of a few megabytes on a medium-end PC. We also demonstrate that a basic compression technique consisting in $q$-gram substitution can significantly reduce the index size (up to 50% of the input text size for the DNA), while still keeping the query time relatively low.

## 1 Introduction

Dictionary string matching (keyword matching, matching in dictionaries), defined as the task of checking if a query string occurs in a collection of strings given beforehand, is a classic research topic. In recent years, increased interest in *approximate* dictionary matching can be observed, where the query and one of the strings from the dictionary may only be similar in a specified sense rather than equal. Approximate dictionary matching is considered a hard problem, since most useful string similarity measures are non-transitive. On the other hand, matching with mismatches (i.e. using a Hamming distance) is also a very desired functionality with applications in, i.a., bioinformatics [21, 22], biometrics [13], cheminformatics [16], circuit design [19], and web crawling [25].

As indexes supporting approximate matching tend to grow exponentially in $k$, the maximum number of allowed errors, it is also a worthwhile goal to design efficient indexes supporting only a small $k$. In this paper, we focus on the problem of dictionary matching with few mismatches (especially one mismatch). Formally, for a collection $\mathcal{D} = \{d_1, \ldots, d_m\}$ of $|\mathcal{D}|$ strings (words) $d_i$ of total length $n$ over

a given alphabet $\Sigma$ (where $\sigma = |\Sigma|$), $I(\mathcal{D})$ is an approximate dictionary index supporting matching with mismatches, if for any query pattern $P$ it returns all strings $d_j$ from $\mathcal{D}$ such that $Ham(P, d_j) \leqslant k$ (Hamming distance). As regards the substrings, they are denoted as $S[i_0, i_1]$ (an inclusive range), and indexes are 0-based.

## 2 Related work

Solutions for approximate dictionary matching can be basically divided into two classes: worst-case space and query time oriented, and heuristical ones. Notable results from the first class include the $k$-errata trie by Cole et al. [11] which is based on the suffix tree and the longest common prefix structure. It can be used in various contexts, including full-text and keyword indexing, as well as wildcard matching. For the Hamming distance and dictionary matching, it uses $O(n + |\mathcal{D}|\frac{(\log|\mathcal{D}|)^k}{k!})$ space and offers $O(m + \frac{(\log|\mathcal{D}|)^k}{k!} \log\log n + occ)$ query time (this also holds for the edit distance but with larger constants). This was extended by Tsur [29] who described a structure similar to the one from Cole et al. with time complexity $O(m+\log\log n+occ)$ (for constant $k$) and $O(n^{1+\varepsilon})$ space for a constant $\varepsilon > 0$. For full-text searching with the Hamming distance, Gabriele et al. [17] provided an index with average search time $O(m+occ)$ and $O(n \log^l n)$ space (for some $l$). Another theoretical work describing the algorithm which is similar to our split index was given by Shi and Widmayer [28], who obtained $O(n)$ preprocessing time and space complexity and $O(n)$ expected search time if $k$ is bounded by $O(m/\log m)$. They introduce the notion of home strings for a given $q$-gram, which is the set of strings in $\mathcal{D}$ that contain the $q$-gram in the exact form (the value of $q$ is set to $|P|/(k+1)$. In the search phase, they partition $P$ into $k+1$ disjoint $q$-grams and use a candidate inspection order to speed up finding the matches with up to $k$ edit distance errors.

On the practical front, Bocek et al. [3] provided a generalization of the Mor–Fraenkel [26] algorithm for $k \geqslant 1$ which is called *FastSS*. To check if two strings $S_1$ and $S_2$ match with up to $k$ errors, we first delete all possible ordered subsets of $k'$ symbols for all $0 \leqslant k' \leqslant k$ from $S_1$ and $S_2$. Then we conclude that $S_1$ and $S_2$ *may* be in edit distance at most $k$ if and only if the intersection of the resulting lists of strings is non-empty (explicit verification is still required). For instance, if $S_1 =$ `abbac` and $k = 2$, then its neighborhood is as follows: `abbac`, `bbac`, `abac`, `abac`, `abbc`, `abba`, `abb`, `aba`, `abc`, `aba`, `abc`, `aac`, `bba`, `bbc`, `bac` and `bac` (some of the resulting strings are repeated and they may be removed). If $S_2 =$ `baxcy`, then its respective neighborhood for $k = 2$ will contain, e.g., the string `bac`, but the following verification will show that $S_1$ and $S_2$ are in edit distance greater than 2. If, however, $Lev(S_1, S_2) \leqslant 2$ (Levenshtein distance), then it is impossible not to have in the neighborhood of $S_2$ at least one string from the neighborhood of $S_1$, hence we will never miss a match. The lookup requires $O(km^k \log(nm^k))$ time (where $m$ is the average dictionary word length) and the index occupies $O(nm^k)$ space. Another practical filter was presented by Karch et al. [20] and it improved on the FastSS method. They reduced space requirements

and the query time by splitting long words (similarly to FastBlockSS which is a variant of the original method) and storing the neighborhood implicitly with indexes and pointers to original dictionary entries. They claimed to be faster than other approaches such as the aforementioned FastSS and a BK-tree [6]. Recently, Chegrane and Belazzougui [9] described another practical index and they reported better results when compared to Karch et al. Their structure is based on the dictionary by Belazzougui for the edit distance of 1 (see the following subsection). An approximate (in the mathematical sense) data structure for approximate matching which is based on the Bloom filter was also described [24].

A permuterm index is a keyword index which supports queries with one wildcard symbol [18]. The idea is store all rotations of a given word appended with the terminating character, for instance for the word `text`, the index would consist of the following permuterm vocabulary: `text$, ext$t, xt$te, t$tex, $text`. When it comes to searching, the query is first rotated so that the wildcard appears at the end, and subsequently its prefix is searched for using the index. This could be for example a trie or any other data structure which supports a prefix lookup. The main problem with the standard permuterm index is its space usage, as the number of strings inserted into the data structure is the number of words multiplied by the average string length. Ferragina and Venturini [15] proposed a *compressed* permuterm index in order to overcome the limitations of the original structure with respect to space. They explored the relation between the permuterm index and the Burrows–Wheeler Transform [7], which is applied to a concatenation of all strings from the input dictionary. They provided a modification of the LF-mapping known from FM-indexes [14] in order to support the functionality of the permuterm index.

### 2.1 The 1-error problem

It is important to consider methods for detecting a single error, since over 80% of errors (even up to roughly 95%) are within $k = 1$ for the edit distance with transpositions [12, 27]. Belazzougui and Venturini [2] presented a compressed index whose space is bounded in terms of the $k$-th order empirical entropy of the indexed dictionary. It can be based either on perfect hashing, having $O(m + occ)$ query time or on a compressed permuterm index with $O(m \min(m, \log_\sigma n \log\log n) + occ)$ time (when $\sigma = \log^c n$ for some constant $c$) but improved space requirements. The former is a compressed variant of a dictionary presented by Belazzougui [1] which is based on neighborhood generation and occupies $O(n \log \sigma)$ space and can answer queries in $O(m)$ time. Chung et al. [10] showed a theoretical work where external memory is used, and their focus is on I/O operations. They limited the number of these operations to $O(1 + m/(wB) + occ/B)$, where $w$ is the size of the machine word and $B$ is the number of words within a block (a basic unit of I/O), with the space of the proposed structure of $O(n/B)$ blocks. In the category of filters, Mor and Fraenkel [26] described a method which is based on the deletion-only 1-neighborhood.

For the 1-mismatch problem, Yao and Yao [30] described the data structure for binary strings with fixed length $m$ with $O(m \log\log |\mathcal{D}|)$ query time and $O(|\mathcal{D}| m \log m)$ space requirements. This was later improved by Brodal and Gąsieniec [4] with a data structure with $O(m)$ query time which occupies $O(n)$ space. This was in turn extended with a structure with $O(1)$ query time and $O(|\mathcal{D}| \log m)$ space in a cell probe model (where only memory accesses are counted) [5]. Another notable example is a recent theoretical work of Chan and Lewenstein [8], who introduced the index with optimal query time (i.e. $O(m/w + occ)$, where $occ$ is the number of pattern occurrences) which uses additional $O(wd \log^{1+\varepsilon} d)$ bits of space (beyond the dictionary of $d$ strings), assuming a constant-size alphabet.

## 3 Our algorithm

The algorithm that we are going to present is uncomplicated and based on the Dirichlet principle, ubiquitous in approximate string matching techniques. We partition each word $d$ into $k+1$ disjoint pieces $p_1, \ldots, p_{k+1}$, of average length $|d|/(k+1)$ (hence the name "split index"), and each such piece acts as a key in a hash table $H_T$. The size of each piece $p_i$ of word $d$ is determined using the following formula: $|p_i| = \lfloor |d|/(k+1) \rceil$ and $|p_{k+1}| = |d| - \sum_{i=1}^{k} |p_i|$, i.e. the piece size is rounded to the nearest integer and the last piece covers the characters which are not in other pieces. This means that the pieces might be in fact unequal in length, e.g., 3 and 2 for $|d| = 5$ and $k = 1$. The values in $H_T$ are the lists of words which have one of their pieces as the corresponding key. In this way, every word occurs on exactly $k+1$ lists. This seemingly bloats the space usage, still, in the case of small $k$ the occupied space is acceptable. Moreover, instead of storing full words on the respective lists, we only store their "missing" prefix or suffix. For instance for the word `table` and $k = 1$, we would have a relation `tab` $\rightarrow$ `le` on one list (i.e. `tab` would be the key and `le` would be the value) and `le` $\rightarrow$ `tab` on the other.

In the case of $k = 1$, we first populate each list with the pieces without their prefix and then with the pieces without the suffix; additionally we store the position on the list (as a 16-bit index) where the latter part begins. In this way, we traverse only a half of a list on average during the search. We can also support $k$ larger than 1 — in this case, we ignore the piece order on a list, and we store $\lceil \log_2(k+1) \rceil$ bits with each piece that indicate which piece of the word (i.e. where is the missing piece) is the list key. Let us note that this approach would also work for $k = 1$, however, it turned out to be less efficient.

As regards the implementation, our focus was on data compactness. In the hash table, we store the buckets which contain word pieces as keys (e.g., `le`) and pointers to the lists which store the missing pieces of the word (e.g., `tab`, `ft`). These pointers are always located right next to the keys, which means that unless we are very unlucky, a specific pointer should already be present in the CPU cache during the traversal. The memory layouts of these substructures are fully contiguous. Successive strings are represented by multiple characters with

a prepended 8-bit counter which specifies the length, and the counter with the value 0 indicates the end of the list. During the traversal, each length can be compared with the length of the piece of the pattern. As mentioned before, the words are partitioned into pieces of fixed length. This means that on average we calculate the Hamming distance for only a half of the pieces on the list, since the rest can be ignored based on their length. Any hash function for strings can be used, and two important considerations are the speed and the number of collisions, since a high number of collisions results in longer buckets, which may in turn have a negative effect on the query time (see Section 4 for further discussion). Figure 1 illustrates the layout of the split index.

The preprocessing stage proceeds as follows:

1. Duplicate keywords are removed from the dictionary $\mathcal{D}$.

The following steps refer to each word $d_i$ from $\mathcal{D}$.

2. The word $d_i$ is split into $k+1$ pieces.
3. For each piece $p_i$: if $p_i \notin H_T$, we create a new list $L_n$ containing the missing pieces $\mathcal{P} = \{p_j : j \in [1, k+1] \wedge j \neq i\}$ and add it to the hash table (we append $p_i$ and the pointer to $L_n$ to the bucket). Otherwise, if $p_i \in T_H$, we append the missing pieces $\mathcal{P}$ to the already existing list $L_l$.

As regards the search:

1. The pattern $P$ is split into $k+1$ pieces.
2. We search for each piece $p_i$ (the prefix and the suffix if $k = 1$): the list $L_l$ is retrieved from the hash table or we continue if $p_i \notin H_T$. Otherwise, we traverse each missing piece $p_j$ from $L_l$. If $|p_j| = |P| - |p_i|$, the verification is performed and the result is returned if $Ham(p_j, P - p_i) \leqslant k$ (where the subtraction sign indicates substring removal).
3. The pieces are combined into one word in order to present the answer.

### 3.1 Complexity

Let us consider the average word length $|d|$, where $|d| = (\sum_{i=1}^{|\mathcal{D}|} |d_i|)/|\mathcal{D}|$. Average time complexity of the preprocessing stage is $O(kn)$, where $k$ is the allowed number of errors, and $n$ is the total input dictionary size (i.e. the length of the concatenation of all words from $\mathcal{D}$, $n = \sum_{i=1}^{|\mathcal{D}|} |d_i|$). This is because for each word and for each piece $p_i$ we can either add the missing pieces to a new list or append them to the already existing one in $O(|d|)$ time (if optimized; let us note that $|\mathcal{D}||d| = n$). We assume that adding a new element to the bucket takes constant time on average, and that the calculation of all hashes takes $O(n)$ time in total. This is true irrespective of which list layout is used (there are two layouts for $k = 1$ and $k > 1$, see the preceding paragraphs). The occupied space is equal to $O(kn)$, because each part appears on exactly $k$ lists and in exactly 1 bucket.

The average search complexity is $O(kt)$, where $t$ is the average length of the list. We search for each of $k+1$ pieces of the pattern of length $m$, and when the list

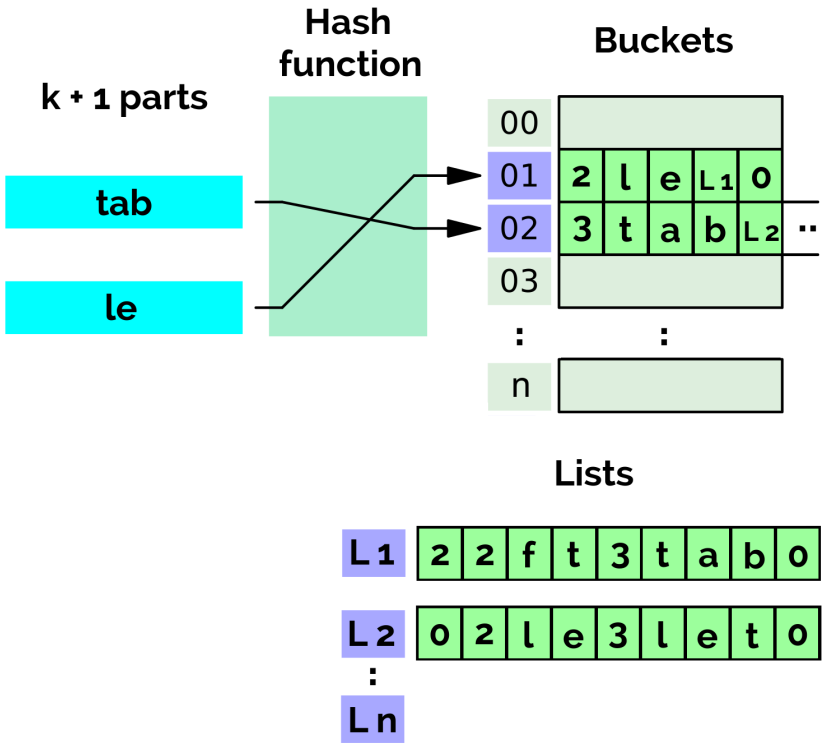


**Fig. 1.** Split index for keyword indexing which shows the insertion of the word `table` for $k = 1$. The index also stores the words `left` and `tablet` (only selected lists containing pieces of these two words are shown), and `L1` and `L2` indicate pointers to the respective lists. The first cell of each list indicates a 1-based word position (i.e. the word count from the left) where the missing prefixes begin ($k = 1$, hence we deal with two parts, namely prefixes and suffixes), and 0 means that the list has only missing suffixes. Adapted from Wikimedia Commons (author: Jorge Stolfi; available at `http://en.wikipedia.org/wiki/File:Hash_table_3_1_1_0_1_0_0_SP.svg`; CC A-SA 3.0).

corresponding to the piece $p_i$ is found, it is traversed and at most $t$ verifications are performed. Each verification takes at most $O(\min(m, |d_{max}|))$ time where $d_{max}$ is the longest word in the dictionary[1], but $O(1)$ time on average. Again, we assume that determining a location of the specific list, that is iterating a bucket, takes $O(1)$ time on average. As regards the list, its average length $t$ is higher when there is a higher probability that two words $d_1$ and $d_2$ from $\mathcal{D}$ have two parts of the same length $l$ which match exactly, i.e. $Pr(d_1[i_1, i_1+l-1] = d_2[i_2, i_2+l-1])$.

[1] Or $O(k)$ time, in theory, using the old longest common extension (LCE) based technique from Landau and Vishkin [23], after $O(n \log \sigma)$-time preprocessing.

Since all words are sampled from the same alphabet $\Sigma$, $t$ depends on the alphabet size, that is $t = f(\sigma)$. Still, the dependence is rather indirect; in real-world dictionaries which store words from a given language, $t$ will be rather dependent on the $k$-th order entropy of the language.

### 3.2 Compression

In order to reduce storage requirements, we apply a basic compression technique. We find the most frequent $q$-grams in the word collection and replace their occurrences on the lists with unused symbols, e.g., byte values $128, \ldots, 255$. The values of $q$ can be specified at the preprocessing stage, for instance $q = 2$ and $q = 4$ are reasonable for the English alphabet and DNA, respectively. Different $q$ values can be also combined depending on the distribution of $q$-grams in the input text, i.e. we may try all possible combinations of $q$-grams up to a certain $q$ value and select ones which provide the best compression. In such a case, longer $q$-grams should be encoded before shorter ones. For example, a word `compression` could be encoded as `#p*s\` using the following substitution list: `com` $\rightarrow$ `#`, `re` $\rightarrow$ `*`, `co` $\rightarrow$ `$`, `om` $\rightarrow$ `&`, `sion` $\rightarrow$ \ (note that not all $q$-grams from the substitution list are used). Possibly even a recursive approach could be applied, although this would certainly have a substantial impact on the query time.

The space usage could be further reduced by the use of a different character encoding. For the DNA (assuming 4 symbols only) it would be sufficient to use 2 bits per character, and for the basic English alphabet 5 bits. In the latter case there are 26 letters, which in a simplified text can be augmented only with a space character, a few punctuation marks, and a capital letter flag. Such an approach would be also beneficial for space compaction, and it could have a further positive impact on cache usage. The compression naturally reduces the space while increasing the search time, and a sort of a middle ground can be achieved by deciding which additional information to store in the index. This can be for instance the length of an encoded (compressed) piece after decoding, which could eliminate some pieces based on their size without performing the decompression and explicit verification.

## 4 Experimental results

Experimental results were obtained on the machine equipped with the Intel i5-3230M processor running at 2.6 GHz and 8 GB DDR3 memory, and the C++ code was compiled with clang version 3.4-1 and run on the Ubuntu 14.04 OS.

One of the crucial components of the split index is a hash function. Ideally, we would like to minimize the average length of the bucket (let us recall that we use chaining for collision resolution), however, the hash function should be also relatively fast because it has to be calculated for each of the $k + 1$ parts of the pattern (of total length $m$). We investigated various hash functions, and it turned out that the differences in query times are not negligible, although the average length of the bucket was almost the same in all cases (relative differences were

smaller than 1%). We can see in Table 1 that the fastest function was the xxhash (available on the Internet under the following link: `https://code.google.com/p/xxhash/`), and for this reason it was used for the calculation of other results.

| Hash function | Query time (µs) |
|---|---|
| xxhash | 0.93 |
| sdbm | 0.95 |
| FNV1 | 0.95 |
| FNV1a | 0.95 |
| SuperFast | 0.96 |
| Murmur3 | 0.97 |
| City | 0.99 |
| FARSH | 1.00 |
| SpookyV2 | 1.04 |
| Farm | 1.04 |

**Table 1.** Evaluated hash functions and search times per query for the English dictionary of size 2.67 MB and $k = 1$. A list of common English misspellings was used as queries, max LF = 2.0.

Decreasing the value of the load factor (LF) did not strictly provide a speedup in terms of the query time, as demonstrated in Figure 2. This can be explained by the fact that even though the relative reduction in the number of collisions was substantial, the absolute difference was equal to at most a few collisions per list. Moreover, when the LF was higher, pointers to the lists could be possibly closer to each other, which might have had a positive effect on cache utilization. The best query time was reported for the maximum LF value of 2.0, hence this value was used for the calculation of other results.

In Table 2 we can see a linear increase in the index size and an exponential increase in query time with growing $k$. Even though we concentrate on $k = 1$ and the most promising results are reported for this case, our index might remain competitive also for higher $k$ values.

| $k$ | Query time (µs) | Index size (KB) |
|---|---|---|
| 1 | 0.51 | 1,715 |
| 2 | 11.49 | 2,248 |
| 3 | 62.85 | 3,078 |

**Table 2.** Query time and index size vs the error value $k$ for the English language dictionary of size 0.79 MB. A list of common English misspellings was used as queries.

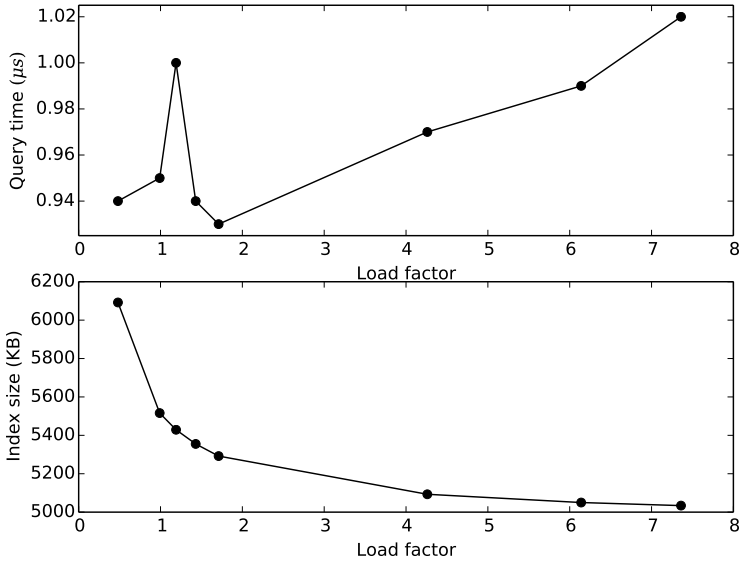


**Fig. 2.** Query time and index size vs the load factor for the English dictionary of size 2.67 MB and $k = 1$. A list of common English misspellings was used as queries. The value of LF can be higher than 1.0 because we use chaining for collision resolution.

$Q$-gram substitution coding provided a reduction in the index size, at the cost of increased query time. $Q$-grams were generated separately for each dictionary $\mathcal{D}$ as a list of 100 $q$-grams which provided the best compression for $\mathcal{D}$, i.e. they minimized the size of all encoded words, $S_E = \sum_{i=1}^{|\mathcal{D}|} |Enc(d_i)|$. For the English language dictionaries, we also considered using only 2-grams or only 3-grams, and for the DNA only 2-grams (a maximum of 25 2-grams) and 4-grams, since mixing the $q$-grams of various sizes has a further negative impact on the query time. For the DNA, the queries were generated randomly by introducing noise into words sampled from dictionary, and their length was equal to the length of the particular word. Up to 3 errors were inserted, each with a 50% probability. For the English dictionaries we opted for the list of common misspellings, and the results were similar to the case of randomly generated queries.

We can see the speed-to-space relation for the English dictionaries in Figure 3 and for the DNA in Figure 4. In the case of English, using the optimal (from the compression point of view, i.e. minimizing the index size) combination of mixed $q$-grams provided almost the same index size as using only 2-grams. Substitution coding methods performed better for the DNA (where $\sigma = 5$) because the sequences are more repetitive. Let us note that the compression provided a higher relative decrease in index size with respect to the original text as the size

of the dictionary increased. For instance, for the dictionary of size 627.8 MB the compression ratio was equal to 1.93 and the query time was still around 100 µs.

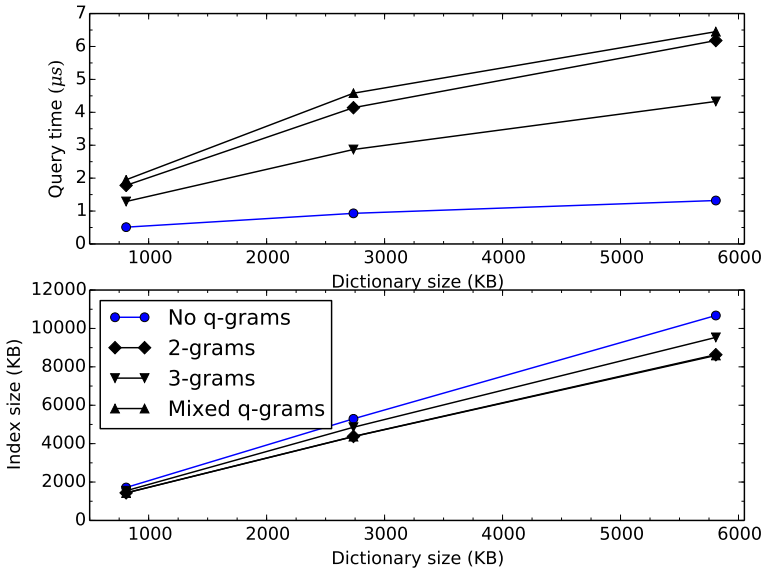


**Fig. 3.** Query time and index size vs dictionary size for $k = 1$, with and without $q$-gram coding. Mixed $q$-grams refer to the combination of $q$-grams which provided the best compression, and for the three dictionaries these were equal to ([2-, 3-, 4-] grams): [88, 8, 4], [96, 2, 2], and [94, 4, 2], respectively. English language dictionaries and the list of common English misspellings were used.

Tested on the English language dictionaries, promising results were reported when compared to methods proposed by other authors. Others consider the Levenshtein distance as the edit distance, whereas we use the Hamming distance, which puts us at the advantageous position. Still, the provided speedup is significant, and we believe that the more restrictive Hamming distance is also an important measure of practical use. The implementations of other authors are available on the Internet (`http://searchivarius.org/personal/software`; `https://code.google.com/p/compact-approximate-string-dictionary/`, from Boytsov and Chegrane and Belazzougui, respectively). As regards the results reported for the MF method and Boytsov's Reduced alphabet neighborhood generation, it was not possible to accurately calculate the size of the index (both implementations by Boytsov), and for this reason we used rough ratios based on index sizes reported by Boytsov for similar dictionary sizes. Let us note that we compare our algorithm with Chegrane and Belazzougui, who report better results when

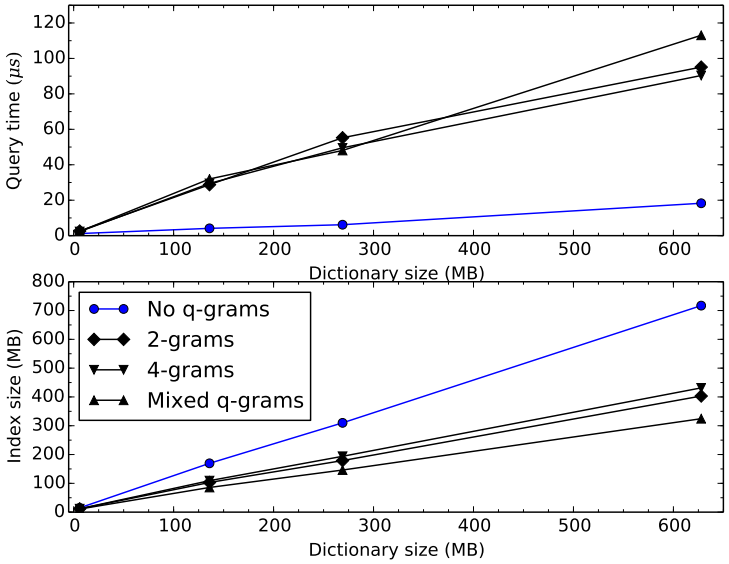


**Fig. 4.** Query time and index size vs dictionary size for $k = 1$, with and without $q$-gram coding. Mixed $q$-grams refer to the combination of $q$-grams which provided the best compression, and these were equal to ([2-, 3-, 4-] grams): [16, 66, 18] (due to computational constraints, they were calculated only for the first dictionary, but used for all four dictionaries). DNA dictionaries and the randomly generated queries were used.

compared to Karch et al., who in turned claimed to be faster than other state-of-the-art methods [9, 20]. We have not managed to identify any practice-oriented indexes for matching in dictionaries over any fixed alphabet $\Sigma$ dedicated for the Hamming distance, which could be directly compared to our split index. The times for the brute-force algorithm are not listed, since they were roughly 3 orders of magnitude higher than the ones presented. Consult Figure 5 for details.

We also evaluated different word splitting schemes. For instance for $k = 1$, one could split the word into two parts of different sizes, e.g., $6 \rightarrow (2, 4)$ instead of $6 \rightarrow (3, 3)$, however, unequal splitting methods caused slower queries when compared the the regular one. As regards Hamming distance calculation, it turned out that a naive implementation (i.e. simply iterating and comparing each character) was the fastest one. The compiler with automatic optimization was simply more efficient than other implementations (e.g., ones based directly on SSE instructions) that we have investigated.

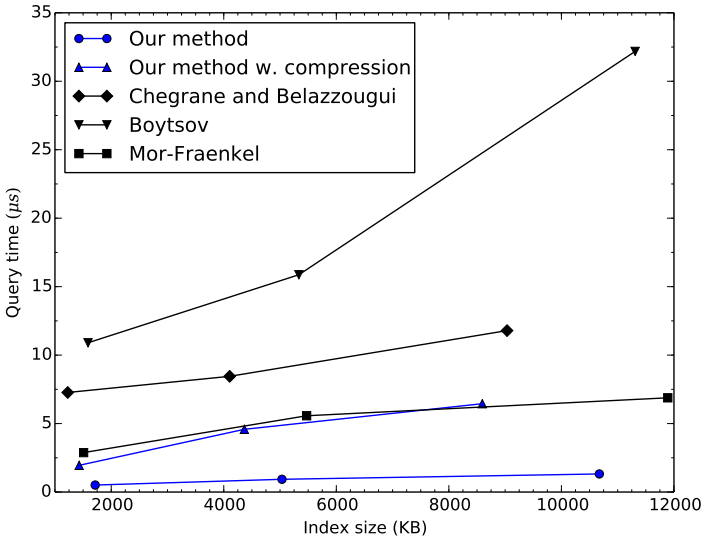


**Fig. 5.** Query time vs index size for different methods. The method with compression encoded mixed $q$-grams. We used the Hamming distance, and the other authors used the Levenshtein distance for $k = 1$. English language dictionaries of size 0.79 MB, 2.67 MB, and 5.8 MB were used as input, and the list of common misspellings was used for queries.

## 5 Conclusions

We have presented an index for dictionary matching with mismatches, which performed best for the Hamming distance of one. Its functionality could be extended by storing additional information in the lists that contain the missing parts of the words. This could be for instance a mapping of words to positions in the document, which would create an inverted index supporting approximate matching.

The algorithm can be sped up by means of parallelization, since access to the index during the search procedure is read-only. In the most straightforward approach we could simply distribute individual words between multiple threads. A more fine-grained variation would be to concurrently operate on parts of the word after it has been split up (the number of parts depending on the $k$ parameter), or we could even access in parallel lists which contain candidate prefixes and suffixes. If we had a sufficient amount of threads at our disposal, these approaches could be combined.

## Appendix A

The following data sets were used in order to obtain the experimental results:

- iamerican — 0.79 MB, English, available from Linux packages
- foster — 2.67 MB, English, available at: `http://www.math.sjsu.edu/~foster/dictionary.txt`
- iamerican-insane — 5.8 MB, English, available from Linux packages
- DNA — 20-mers extracted from the genome of Drosophila melanogaster (available at: `http://flybase.org/`), sizes: 6.01 MB, 135.89 MB, 262.78 MB, and 627.80 MB
- A list of common English misspellings — 44.2 KB (4,261 words), available at: `http://en.wikipedia.org/wiki/Wikipedia:Lists_of_common_misspellings/For_machines`

## References


1. D. Belazzougui. Faster and space-optimal edit distance "1" dictionary. In G. Kucherov and E. Ukkonen, editors, *Combinatorial Pattern Matching, 20th Annual Symposium, CPM 2009, Lille, France, June 22-24, 2009, Proceedings*, volume 5577 of *Lecture Notes in Computer Science*, pages 154–167. Springer, 2009.
2. D. Belazzougui and R. Venturini. Compressed string dictionary look-up with edit distance one. In J. Kärkkäinen and J. Stoye, editors, *Combinatorial Pattern Matching - 23rd Annual Symposium, CPM 2012, Helsinki, Finland, July 3-5, 2012. Proceedings*, volume 7354 of *Lecture Notes in Computer Science*, pages 280–292. Springer, 2012.
3. T. Bocek, E. Hunt, B. Stiller, and F. Hecht. Fast similarity search in large dictionaries. Technical Report ifi-2007.02, Department of Informatics, University of Zurich, 2007.
4. G. S. Brodal and L. Gąsieniec. Approximate dictionary queries. In *Combinatorial Pattern Matching*, pages 65–74. Springer, 1996.
5. G. S. Brodal and S. Venkatesh. Improved bounds for dictionary look-up with one error. *Information Processing Letters*, 75(1):57–59, 2000.
6. W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Communications of the ACM*, 16(4):230–236, 1973.
7. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, CA, 1994.
8. T. M. Chan and M. Lewenstein. Fast string dictionary lookup with one error. In F. Cicalese, E. Porat, and U. Vaccaro, editors, *Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 - July 1, 2015, Proceedings*, volume 9133 of *Lecture Notes in Computer Science*, pages 114–123. Springer, 2015.
9. I. Chegrane and D. Belazzougui. Simple, compact and robust approximate string dictionary. *Journal of Discrete Algorithms*, 28:49–60, 2014.
10. C. Chung, Y. Tao, and W. Wang. I/O-efficient dictionary search with one edit error. In E. S. de Moura and M. Crochemore, editors, *String Processing and Information Retrieval - 21st International Symposium, SPIRE 2014, Ouro Preto, Brazil, October 20-22, 2014. Proceedings*, volume 8799 of *Lecture Notes in Computer Science*, pages 191–202. Springer, 2014.

11. R. Cole, L.-A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 91–100. ACM, 2004.
12. F. J. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.
13. G. I. Davida, Y. Frankel, and B. J. Matt. On enabling secure applications through off-line biometric identification. In *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*, pages 148–157. IEEE, 1998.
14. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.
15. P. Ferragina and R. Venturini. The compressed permuterm index. *ACM Transactions on Algorithms (TALG)*, 7(1):10, 2010.
16. D. R. Flower. On the properties of bit string-based measures of chemical similarity. *Journal of Chemical Information and Computer Sciences*, 38(3):379–386, 1998.
17. A. Gabriele, F. Mignosi, A. Restivo, and M. Sciortino. Indexing structures for approximate string matching. In *Algorithms and Complexity*, pages 140–151. Springer, 2003.
18. E. Garfield. The permuterm subject index: An autobiographical review. *Journal of the American Society for Information Science*, 27(5):288–291, 1976.
19. P. Girard, C. Landrault, S. Pravossoudovitch, and D. Severac. Reduction of power consumption during test application by test vector ordering. *Electronics Letters*, 33(21):1752–1754, 1997.
20. D. Karch, D. Luxen, and P. Sanders. Improved fast similarity search in dictionaries. In *String Processing and Information Retrieval*, pages 173–178. Springer, 2010.
21. S. Kurtz, J. V. Choudhuri, E. Ohlebusch, C. Schleiermacher, J. Stoye, and R. Giegerich. REPuter: the manifold applications of repeat analysis on a genomic scale. *Nucleic Acids Research*, 29(22):4633–4642, 2001.
22. G. M. Landau, J. P. Schmidt, and D. Sokol. An algorithm for approximate tandem repeats. *Journal of Computational Biology*, 8(1):1–18, 2001.
23. G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of algorithms*, 10(2):157–169, 1989.
24. U. Manber and S. Wu. An algorithm for approximate membership checking with application to password security. *Information Processing Letters*, 50(4):191–197, 1994.
25. G. S. Manku, A. Jain, and A. Das Sarma. Detecting near-duplicates for web crawling. In *Proceedings of the 16th international conference on World Wide Web*, pages 141–150. ACM, 2007.
26. M. Mor and A. S. Fraenkel. A hash code method for detecting and correcting spelling errors. *Communications of the ACM*, 25(12):935–938, 1982.
27. J. J. Pollock and A. Zamora. Automatic spelling correction in scientific and scholarly text. *Communications of the ACM*, 27(4):358–368, 1984.
28. F. Shi and P. Widmayer. Approximate multiple string searching by clustering. *Genome Informatics*, 7:33–40, 1996.
29. D. Tsur. Fast index for approximate string matching. *Journal of Discrete Algorithms*, 8(4):339–345, 2010.
30. A. C. Yao and F. F. Yao. Dictionary look-up with small errors. In *Combinatorial Pattern Matching*, pages 387–394, 1995.